We just saw how the state of input switches could be read into the computer although the way that this data can be used remains a mystery. Obviously the computer must be able to examine this data and act upon it. That is the subject of this chapter, and during the following discussions we will introduce the subject of jump and conditional jump instructions. We will temporatily leave the subject of switches, but shall in the next chapter marry the concept of data inputs (switches) and conditional jumps (decision making) to expand the program for reading and reacting to data inputs.

Jumping Around. During chapter three there was much discussion over the way that the program counter functioned in the instruction fetch cycle. You will remember that at that time we pointed out that it was the contents of the program counter that were sent out over the address bus to the memory in order to specify which memory location we wished to read or to load. As each instruction was executed the program counter was incremented by one, two or three so that it pointed to the next instruction to be executed in the program. While programs are normally executed in numerical ascending order of the addresses containing the instruction, 0100H, 0101H, 0102H, etc., the need frequently occurs for a jump to an instruction out of the normal sequence. Thus in the program of the last chapter, it is possible to place the FFH somewhere in memory other than at location 0103H and jump to it after reading the input port into the accumulator.

A jump occurs in response to an instruction called JUMP. Its mnemonic or abbreviation is JMP and it is always followed with four hex digits that represent the address that is the destination of the jump. Thus JMP 200H will cause the computer to jump to location 0200H and resume program execution at that point. The mechanism for doing this is as follows. When the program counter is incremented to the next location in memory it initiates an instruction fetch cycle. The

data at the location specified by the contents of the program counter are fetched to the CPU and executed. If the instruction is one that requires more data, that data is fetched from the memory in subsequent fetch cycles. When all of the data is present in the CPU, the instruction is executed. If the instruction is one that requires more data, that data is fetched from the memory in subsequent fetch cycles. When all of the data is present in the CPU, the instruction is executed.

The hex code for the JMP instruction is C3H. It must be followed by four hex digits representing the address to which the jump is to occur. As in the case of the LDA instruction, the address must be loaded into the memory with the two least significant digits first, followed by the two most significant digits. We will use the program of the last chapter to try the JMP instruction. Place an FFH program stopper at location 0200H. This will then become our jump destination, and we will be able to tell from the displays whether the jump occured.

Enter:	See Displayed:
CLR	
0 2 0 0 NXT	
F F	
NXT	

We should also be sure that the program from the last chapter is still loaded in the memory. Check it and reload any locations that may have changed.

Enter:	See Displayed:
DCM 0 1 0 0 NXT	
NXT	
NXT	

At this point insert the new JMP instruction erasing the FFH that was in location 0103H that provided the original program stopper.

Enter:	See Displayed:
NXT	
C 3	
NXT 0 0	
NXT 0 2	
NXT	

The complete program is now loaded; a summary appears below.

```
0100    3A      LDA     C000H       ;Load the accumulator with the
0101    00                          ;contents of the input port C000H.
0102    CO                          ;
0103    C3      JMP     0200H       ;Jump to location 0200H.
0104    00                          ;
```

```
0105    02
0200    FF    RST   7          ;Stop the program and enter STEP.
```

Reset the DIP switches to the ON or CLOSED position and execute the program. This had previously caused the accumulator to be loaded with 83H. Will it again?

Enter:	See Displayed:
CLR	
EXC	

Apparently the jump occurred properly. Check the accumulator.

Enter:	See Displayed:
	
DCR	

What's more, inserting the jump into the program did not change the way the accumulator was loaded with the switch data. Let's go through it one step at a time.

When we press EXC , the contents of the program counter will be sent to the memory to fetch the next instruction. That instruction will be returned by the memory to the CPU over the data bus and stored inside the CPU in a special register called the instruction register (this register is not directly accessible to the user and is not displayed in the DCR mode). There the instruction is decoded by the CPU which takes action depending on which instruction has been returned by

the memory. In the case of both LDA and JMP, the CPU recognized that these instructions require more data, namely the four hex digits that represent the address needed by the instruction. The CPU saves the instruction code in the instruction register for safe-keeping, but it increments the program counter and sends the contents out on the address bus to fetch more data. As that data is returned to the CPU, the CPU recognizes that this is the least significant two digits of the address needed by the instruction in the instruction register. Actually the CPU will interpret whatever it finds at that location as the two digits of the address; it is up to the person writing the program to be sure that the address is correctly stored at that location or an error will result. In any event, the CPU interprets the incoming data as the stuff addresses are made of and stores it in a temporary holding register. Notice the instruction in the instruction register is not disturbed or replaced by the incoming data. The program counter is then incremented again and another two hex digits fetched from the memory. These are added to the first two in the temporary register and a four digit address results. In the case of the LDA instruction, this address is then sent to the memory and the data that is returned is loaded into the accumulator.

In the case of the JMP instruction, however, something totally different happens. The address that is assembled in the temporary register is loaded into the program counter right on top of the address that was already there. The original address in the computer is completely destroyed; it has been replaced by the jump address specified in the JMP instruction. During the next instruction fetch cycle it will be this new address that is sent out over the address bus and is subsequently incremented. The program execution has, in fact, jumped to a new location and subsequent instructions will be executed beginning at that point.

Enter:	See Displayed:
	
NXT	
NXT	
NXT	
NXT	
NXT	
NXT	
NXT	
NXT	
NXT	

As we had suspected, the program counter now points to the next instruction to be executed out of 0201H.

The ability of the programmer to make the computer jump to a different point in the memory and begin executing instructions from there is a powerful tool. We will not, however appreciate the implications of this ability until we study another variety of the jump instruction, the conditional jump. Therein is the key to the decision making process of the computer.

The Conditional Jump. The concept of the conditional jump is very straight forward. Unlike the JMP instruction which causes a jump each and every time it is executed, the conditional jump may or may not cause a jump depending on whether or not certain conditions are true. If the jump occurs the instruction is executed in exactly the same way as JMP. If the condition for jump is not met, the jump does not occur and the computer simply ignores the instruction. The program counter is incremented right on past the conditional jump instruction and the next instruction in the sequence is executed just as though the conditional jump did not even appear in the program.

Let's build up a little program to try out one of the conditional jumps, the JNZ or Jump If Not Zero instruction. We will use this to test one of the registers to see whether or not it contains zero. The register that we will use is the E register, up to now only displayed briefly in passing through the DCR mode. It is the first of the general purpose working registers with which we will experiment; eventually all of the registers will be explored. For purposes that will become clear in a moment, we wish to decrement the contents of the register before we test it with the JNZ instruction. This is done with an instruction much like the INR A instruction that we used in Chapter Three. DCR E, or Decrement Register E, causes the contents of the E register to be reduced or decremented by one. In so doing, we set the stage for the JNZ instruction.

If after decrementing E, the register contains any non-zero number, the JNZ will cause a jump to some specified address. Check to be sure the program stopper at location 0200H is still in place and then load the following program beginning at location 0300H. The hex code for DCR E is 1DH and that of JNZ is C2. The jump address of JNZ must be loaded into the memory with the least two significant digits first. We will use 0200H as the jump address.

Enter:	See Displayed:
CLR	
DCM 0 2 0 0 NXT	
DCM 0 3 0 0 NXT	
1 D	
NXT C 2	
NXT 0 0	
NXT 0 2	
NXT F F	
NXT	

Before we can execute this program we must set the registers up. We will want to load the E register with 04H. If the EXC key is pressed the computer will begin executing instructions at wherever the program counter is set, which, since we pressed CLR , will be at 0100H. That is not where our program starts, so we will have to set the program counter to 0300H via the DCR mode.

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT NXT	
0 4	
NXT NXT NXT NXT	
0 3 0 0	
NXT	

The program is summarized below and a flow chart appears in Fig. 5-1.

```
0200    FF      RST     7       ;Program stopper and jump destination.

0300    1D      DCR     E       ;Decrement the E register and Jump, if
0301    C2      JNZ     0200H   ;not zero, the location 0200H. Otherwise,
0302    00                      ;stop at location 0304H.
0303    02                      ;
0304    FF      RST     7       ;Optional program stopper.
```

Execute the program and note the address that appears on the displays when the program stopper is reached.

Enter:	See Displayed:
EXC	
DCR	
NXT NXT NXT NXT NXT	

Apparently the E register was decremented, since it now contains 03H, and the JNZ caused the jump to occur to location 0200H where one of our program stoppers resides. Let's single-step through it. First, reset the program counter to 0300H.

Enter:	See Displayed:
	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
STEP	

Has the E register really been decremented? Ask it and find out.

Enter:	See Displayed:
	F0301-C2
DCR	uA------
NXT NXT NXT NXT NXT	uE-----02

So E register is one less than it was a second ago. The next instruction to be executed is the JNZ at 0301H. Since E obviously does not now contain 00H, the condition for jump is met and jump to location 0200H should occur.

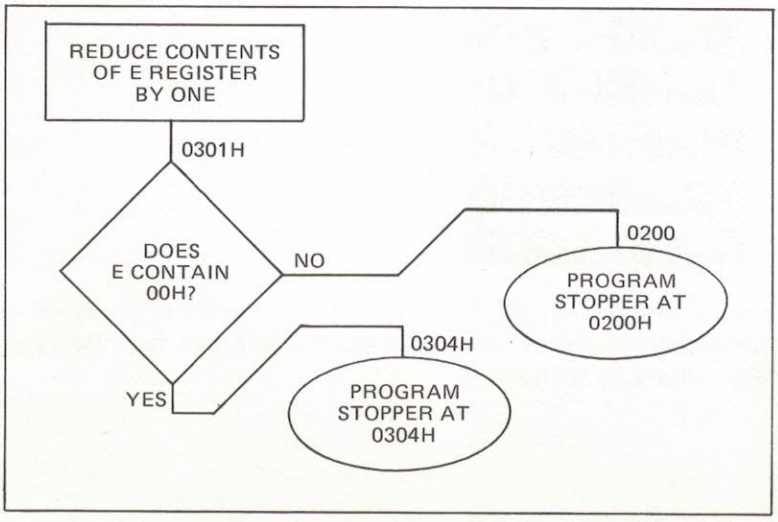


Fig.5-1. The program flow chart for the JNZ instruction. The JNZ at location 0301H represents a decision point for the computer, shown as a diamond in the diagram. If the jump does not occur, the program is said to fall through to location 0304H, represented by the vertical line from the bottom of the diamond. If the jump does occur the program emerges from the diamond at one of the side apexes.

Enter:	See Displayed:
	
STEP	

Reset the program counter to 0300H and try to program again. This time the E register contains 02H so the same result should occur. After being decremented, E will contain 01H.

Enter:	See Displayed:
	
DCR NXT NXT NXT NXT NXT	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
EXC	

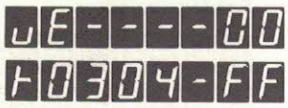
The jump occurred as expected, and a quick check with DCR will reveal that the E register does indeed contain 01H. If we repeat the program once more, we should see the JNZ test fail. Reset the program counter to 0300H and STEP your way through it.

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT NXT	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
STEP	

The E register should now contain 00H.

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT NXT	

This time the JNZ test should fail! Since the E register does now for the first time contain zero, the condition for the jump no longer exists and the JNZ instruction should be passed over and ignored.

Enter:	See Displayed:
	
STEP	

There it is. The jump condition was not met and the program fell through to the program stopper at location 0304H. Your ia7301 just made its first decision, that the E register now for the first time contained zero.

The Loop. With just a few minor changes to our program we can make it more interesting. Up to now, for example, we have had to reset the program counter after each pass through the decision point. But what would happen if the jump address of the JNZ instruction were to be part of the program instead of 0200H which is out of the main stream of the program. Let's try making the jump address 0300H which means that each time the jump occurs it will be back to the DCR E instruction so that on each pass the register is decremented by one. This will continue until finally the E register contains zero and the program falls through to the program stopper at location 0304H. Let's try it.

First, we'll have to modify the jump address in the JNZ instruction.

Enter:	See Displayed:
	
DCM	
0 3 0 0 NXT	

NXT → 0301-C2

NXT → 0302-00

NXT 0 3 → 0303-03

NXT → 0304-

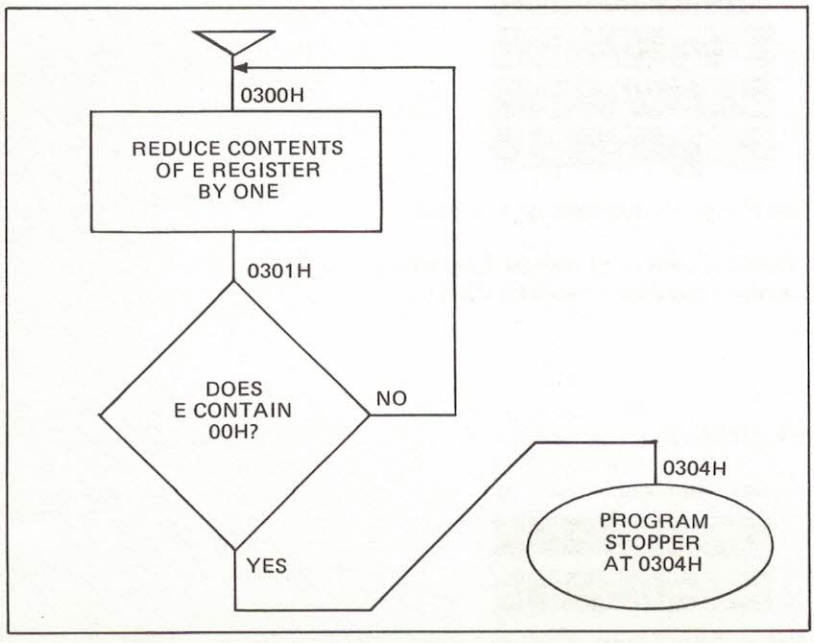


Fig. 5-2. If we change the jump address of the JNZ at 0301H to 0300H, a loop results. The computer will execute this loop over and over, decrementing the E register on each pass, until finally, E=0 and the program falls through to the stopper at 0304H.

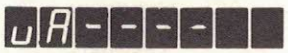
Now set the E register to 04H and the program counter to 0300H.

Enter:	See Displayed:
	
DCR NXT NXT NXT NXT NXT	
0 4	
NXT NXT NXT NXT	
0 3 0 0	
NXT	

The program is summarized below and in the flow diagram of Fig. 5-2.

```
0300    1D    DCR    E        ;Reduce contents of register E by one.
0301    C2    JNZ    0300H    ;Jump if not Zero to address 0300H.
0302    00                    ;
0303    03                    ;
0304    FF                    ;
```

Execute it and let's see if the loop idea is practical.

Enter:	See Displayed:
	
EXC	

Well if the computer went through those four loops it didn't take long to do it. Reset the program counter and the E register and try it again, this time single-stepping through it.

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT NXT	
0 4	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
STEP	
STEP	
STEP	
STEP	
STEP	
STEP	
STEP	
STEP	

The single-step mode shows that the computer in fact does go around the loop four times falling through on the fourth. Apparently we see no delay because the computer executes the program so fast that it happens before we can see it. Let's try it again using FFH in the E register to make the loop FFH times (256 if you're still thinking in decimal).

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT NXT F F	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
EXC	

Still couldn't see any lag between pressing EXC and seeing the displays come up in the STEP mode? Well, we'd better figure out a way to make the delay preceptible since timing functions are common problems faced by microcomputer programmers. Delays ranging from a few milliseconds to seconds or even minutes appear in most practical programs. If loading the largest hexadecimal number possible, FFH, into the register doesn't cause a reasonable delay, how can we extend the delay? We could repeat the loop over and over again, picking up a few microseconds for each loop added to the program. But this could run into hundreds or even thousands of instructions for long delays.

The answer is to place a loop within a loop so that the first must be executed 256 times to decrement the other once. Combining the two loops in this fashion can increase the number of passes to over 65,000. The technique is probably best understood by examining the flow diagram in Fig. 5-3.

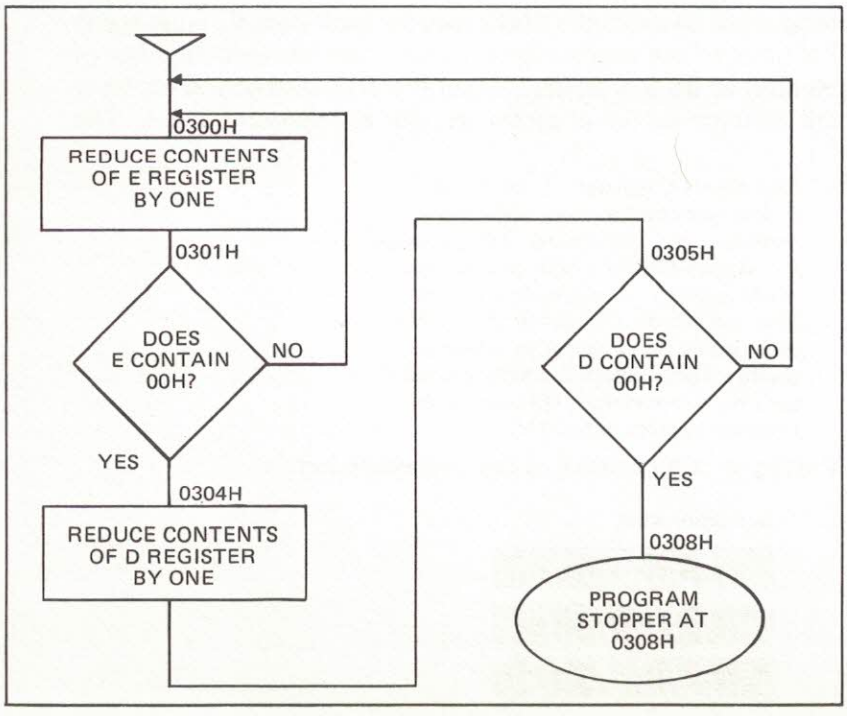


Fig. 5-3. By combining the loop decrementing the E register within a second loop decrementing the D register, the delay can be extended by a factor of 256.

You'll recognize the original loop that decrements the E register and loops until it is zero. Immediately following this loop is a second that acts on the D register. Every time the E-loop falls through, it causes the D loop to be entered which in turn decrements D once. Assuming D originally contained a large number, decrementing it once will undoubtedly not cause it to be reduced to zero. A second JNZ instruction in this loop causes the program to return to the beginning of the first loop. Thus the E loop must be executed 256 times for each pass through the D loop. If E and D both contained FFH prior to the beginning of the program the total number of passes through the E loop will be 256x256 or 65,536 passes. When the JNZ instruction of the D loop finally fails, the program will fall through to the program stopper at location 0308H. The program is summarized below:

```
0300    1D    DCR   E       ;Decrement E register. Then Jump if
0301    C2    JNZ   0300H   ;E does not contain zero. When E
0302    00                  ;contains zero, the program falls through
0303    03                  ;to location 0304H which decrements
0304    15    DCR   D       ;the D register. This sets up a second
0305    C2    JNZ   0300H   ;loop around the D register. A second
0306    00                  ;JNZ causes the program to return to
0307    03                  ;the first loop. When D finally contains
0308    FF    RST   7       ;zero the program falls through to the
                            ;program stopper at 0308H.
```

We enter the program and initialize the registers by following the procedure below.

Enter:	See Displayed:
DCM 0 3 0 0 NXT	
NXT	
NXT	

NXT	
NXT 1 5	
NXT C 2	
NXT 0 0	
NXT 0 3	
NXT F F	
NXT	
DCR NXT NXT NXT NXT	
F F	
NXT	
F F	
NXT NXT NXT NXT	
0 3 0 0	
NXT	

All that remains is to execute the program. Watch to see if you can detect any delay between the time you press EXC and the displays come up in the STEP mode.

Enter:	See Displayed:
	
EXC	

Here for the first time you should have seen a delay. Try it again and try to get a reading on how long the delay is.

Enter:	See Displayed:
	
DCR	
NXT NXT NXT NXT	
F F	
NXT	
F F	
NXT NXT NXT NXT	
0 3 0 0	
NXT	
EXC	

Hard to say exactly how long the delay is, but it is certainly something less than a second. Here is proof of something that the novice programmer finds hard to accept. The computer can execute over 65,500 program loops in less than a second! With this kind of speed at our disposal all sorts of things become possible.

Subroutine Calls. Now that we are starting to get delays that are visible, how can we extend the delay even beyond the range of the double loop? One way would be to just repeat the program over again so that a total of 135,000 passes are executed. This should result in a delay of over one second. Rather than do the obvious we will take advantage of the situation to illustrate one of the most useful of all programming techniques.

Because all programs contain portions that must be used over and over again, most computers have a special feature built into them to facilitate repeating program segments. This feature, which requires that the desired segment be set up as a subroutine, allows the segment to be executed many times without having to repeat the actual instructions. This saves memory locations since the instructions will only be loaded once, and in the world of microcomputers, memory is money. The goal of any programmer should always be to find a way to make his programs shorter so that they require less memory.

Use of subroutines is generally done by setting up the program in two different parts of the memory. The main program stream is in one part and the subroutines are in the other. The main program is executed until a point is reached where the user desires to execute the segment that is to be later repeated. A special instruction, termed a CALL, is executed which causes a jump to the subroutine. Since a program will typically contain many subroutines the CALL instruction will contain the address of the first instruction in the subroutine in much the same way that the JMP instruction contains a jump address. When the subroutine call is executed, the contents of the program counter are replaced with the address contained in the CALL. Program execution con-

tinues at that point and the subroutine is executed. Some way must be found to get the program execution back into the main program stream, however, and a special register in the CPU is dedicated to just this purpose. So far the description of the CALL instruction is exactly the same as that of the JMP insturction. Both cause the program counter to be loaded with the address contained in the instruction. Both cause program execution to continue at the first instruction in the remote segment. The JMP, however, takes the program to this point forever; there is no return provision for automatically getting the program execution back in to the main program stream. If a return occurs at all it is only because the programmer concluded the remote segment with a second JMP instruction which returns the execution back to the main program. This second JMP instruction means that the program segment can only be used once in any program since trying to jump to it from any other point in the program will always cause a return to the same point.

The difference between the two instructions is in the use by the CALL instruction of a special register called the Stack Pointer. While the former contents of the program counter are lost during the process of loading the JMP address, the CALL instruction saves the original contents of the program counter before loading the new address. We will pass over the mechanism for saving the original program counter contents for now and merely remark that the saving process takes place AFTER the program counter is incremented. At the end of the program segment we place a special terminating instruction called Return, RET. This causes the program counter to be reloaded with the original contents (incremented by one) which causes program execution to resume at the point we left off. The effect is as though the computer took a time out to execute the program segment and then went back to the original program.

We can set the program that we have been working on up as a subroutine just by adding the RET, C9H, at the end of the program replacing the FFH program stopper that we had originally at that point.

Enter:	See Displayed:
	0309-__
DCM 0 3 0 8 NXT	0308-FF
C 9	0308-C9
NXT	0309-__

All right, that takes care of the subroutine, but what about the main program. Since the program counter is automatically set to 0100H by the CLR key we can simplify the start-up process somewhat by loading the main program at 0100H. All that will remain is to initialize the D and E registers each to FFH to fulfill their timing function. The machine code for CALL is CDH and must always be followed by the address of the subroutine, least two significant digits first.

Enter:	See Displayed:
	0309-__
DCM 0 1 0 0 NXT	0100-__
C D	0100-Cd
NXT 0 0	0101-00
NXT 0 3	0102-03
NXT C D	0103-Cd

NXT 0 0	0104-00
NXT 0 3	0105-03
NXT C D	0106-CD
NXT 0 0	0107-00
NXT 0 3	0108-03
NXT F F	0109-FF
NXT	010A-__

The program is summarized below and in the diagrams of Fig. 5-4 and 5-5.

```
0100    CD      CALL    DELAY   (0300H)     ;The subroutine located at
0101    00                                  ;location 0300H which causes
0102    03                                  ;a half second delay is called
0103    CD      CALL    DELAY   (0300H)     ;three times for a total delay
0104    00                                  ;of about two seconds.
0105    03                                  ;
0106    CD      CALL    DELAY   (0300H)     ;
0107    00                                  ;
0108    03                                  ;
0109    FF      RST     7                   ;Program Stopper.

0300    1D      DCR     E                   ;Delay subroutine.
0301    C2      JNZ     0300H               ;
0302    00                                  ;
0303    03                                  ;
```

```
0304    15      DCR     D       ;
0305    C2      JNZ     0300H   ;
0306    00                      ;
0307    03                      ;
0308    C9      RET             ;
```

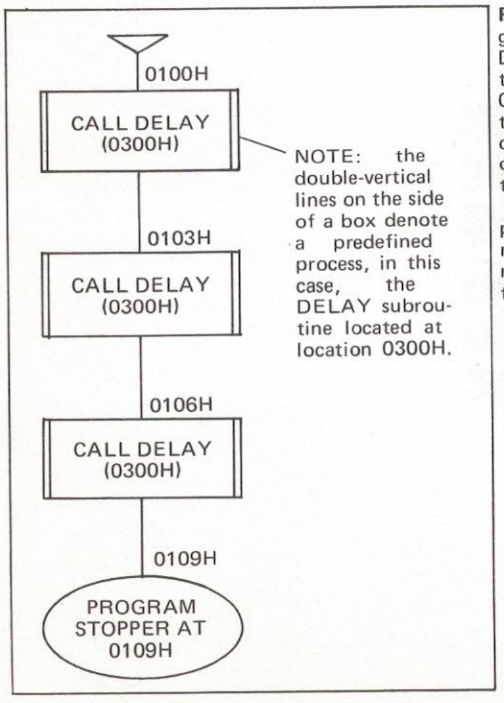


Fig. 5-4. This program CALLS the DELAY subroutine at location 0300H three times to extend the total delay to 2 seconds. Each time the subroutine is completed, program control is returned to the apropriate place in the program.

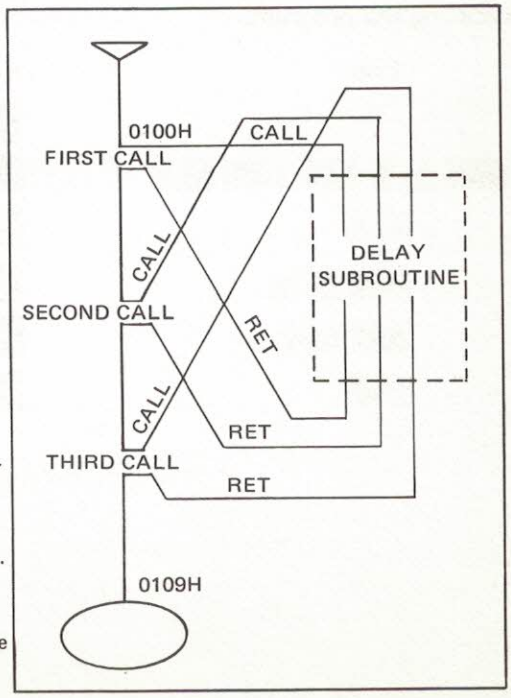


Fig. 5-5. Each time the DELAY subroutine is CALLed, the return address is saved by the CPU. This allows the computer to return to the correct spot in the main program stream.

As we can see in the summary we CALL the delay subroutine three different times in the program; each is good for a half second or so. Each time the subroutine is called the return address is saved so that when the RET instruction in the subroutine is executed, program execution will return to the correct point in the main program. Let's initialize the D and E registers and try executing the program.

Enter:	See Displayed:
	
DCR NXT NXT NXT NXT	
F F	
NXT F F	
NXT CLR	
EXC	

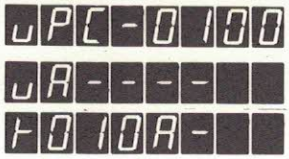
As we had expected, the new program produced a delay of almost two seconds; apparently the system of call subroutines works. Let's try it once more and this time we want you to watch the three LEDs located at the left-hand bottom corner of the board. These are provided because three of the modes of operation may not provide displays on the regular seven-segment displays that up till now we have been using. Thus if the computer is reading or writing programs into a tape recorder no displays will be produced, and you will not be able to determine what the computer is doing. However the three LEDs indicate when the system is reading or writing tape and when the operation is completed. Along this same line, many of the programs that you write and execute will not use the regular displays. The third LED, EXECUTE, provides a visible indication that the computer is executing your program. Execute the program once more and watch the EXECUTE LED.

Enter:	See Displayed:
	
DCR NXT NXT NXT NXT	
F F	
NXT F F	
NXT NXT NXT NXT	

0 1 0 0

NXT

EXC



You should have seen the EXECUTE LED light for the two seconds that the displays were blanked. This is your indication that even though the seven-segment displays are blank, the computer is executing a program.

The Role of the Stack Pointer. The computer displays the stack pointer for you during the DCR mode, but we have always breezed right past it in our rush to get to the program counter. Since the stack pointer is the key to saving addresses that are used by the computer when executing a RET instruction, let's stop and examine the action of this register.

We have already pointed out that the monitor program sets the program counter to 0100H whenever CLR is pressed. At the same time it also sets the stack pointer to 0100H. The stack pointer "points" to the location in memory that will be used to store the return address. Does this mean that the return address will be stored at location 0100H? Because if it does there's going to be quite a conflict because that's where the first instruction of our program is located! We will see in a moment that the conflict is neatly resolved by the fact that the computer decrements the stack pointer once before storing the return address in the memory. Be sure to keep in mind that the stack pointer does not contain the return address itself. Instead, it contains the address of the memory location where the return address is stored. Let's set up and execute the program once more, this time single-stepping our way through it to watch the stack pointer perform its function during the CALLing and RETurning from subroutine calls.

Enter:	See Displayed:
DCR NXT NXT NXT NXT	
F F	
NXT F F	
NXT NXT NXT	

Notice that the stack pointer contains 0100H. Apparently it remains unchanged from the last time we pressed CLR . We will be watching it closely during the execution of our program. Now set the program counter to 0100H to start the program.

Enter:	See Displayed:
NXT 0 1 0 0	
NXT	

In our earlier remarks we indicated that the stack pointer would be decremented by one prior to saving the return address in the memory. If we decrement the present contents of the stack pointer, 0100H, it will contain 00FFH. However, since an address is made up of four hex digits it will take two memory locations to hold the entire return address. The stack pointer will have to be decremented once more and the second two digits stored in memory location 00FEH. To make it easy to see this action occur, we will load 00FEH and 00FFH each with the data FFH. That way any change in the contents of those two locations will be immediately obvious.

Enter:	See Displayed:
DCM 0 0 F E NXT	n00FE-..
F F	n00FE-FF
NXT F F	n00FF-FF
NXT	n0100-Cd

All right we have set location 00FEH and 00FFH to both contain FFH. The very next location, 0100H, contains the first instruction of the program, the CDH that corresponds to the CALL instruction. Thus the memory locations that will be used to store the return address are immediately below the beginning of the program proper. Execute the first instruction.

Enter:	See Displayed:
STEP	P0300-1d

We have executed the CALL instruction that occupies locations 0100H, 0101H and 0102H. The CALL causes a jump to location 0300H where the delay program subroutine is stored, and it is that location that is now displayed on the LEDS. During the execution of the CALL instruction, the return address should have been stored in the memory. Since the CALL occupied locations 0100H, 0101H and 0102H, the next instruction to be executed out of the main program stream will be at location 0103H. The return address to be saved is, therefore, 0103H, and it should now appear in memory locations 00FEH and 00FFH. The stack pointer should also be pointing to these locations so that the CPU will know where to find the return addresses when it executed the RET instruction.

Enter:	See Displayed:
DCM 0 0 F E NXT	
NXT	

There's the return address, 0103H. Notice that it is stored in memory with the most significant two digits in the higher address, that is the 01H is stored in 00FFH and the 03H is stored in 00FEH. Now that the return address has been stored in the memory, how will the computer find the return address when the time comes to return from the subroutine? Let's examine the stack pointer.

Enter:	See Displayed:
DCR NXT NXT NXT NXT	
NXT NXT NXT NXT	

The stack pointer contains the address of the lower memory location holding the least significant two digits of the return address. Since the stack pointer contained 0100H prior to the subroutine call, it has been apparently decremented by two. When the RET instruction that concludes the subroutine is exited, it will use the address contained in the stack pointer to retrieve the return address, 0103H, from the memory. If we try to step our way through the program, we'll have a while to wait before the RET can be exited, since the subroutine uses a loop with 256 passes. That loop, you will remember, decrements the D and E register pair on each pass until the registers contain zero. The looping action will then end and the computer will execute the RET instruction and leave the subroutine to return to the main program stream.

In order to cut short the subroutine so that we can STEP through the execution of the RET instruction, we will use the DCR mode to preset both the D and E registers to 01H. This will have the effect of making the computer exit the loop on the first pass. This is because the first instruction in the subroutine, DCR E, will decrement the E register so that it contains 00H, and since the D register has been set so that it too contains 01H, both JNZ tests will fail.

Enter:	See Displayed:
	
NXT NXT NXT NXT NXT NXT	
0 1	
NXT	
0 1	
NXT	
STEP	
STEP	
STEP	
STEP	

There's the RET instruction. The next depression of STEP should cause the RET at 0308H to be executed and program execution to pick up at 0103H.

Enter:	See Displayed:
STEP	

Just as we predicted! But let's check the stack pointer.

Enter:	See Displayed:
DCR NXT NXT NXT NXT	
NXT NXT NXT NXT	

In the process of executing the RET instruction the contents of the stack pointer are incremented twice, since we just saw, its 00FEH contents have been changed to 0100H. This is done by using the contents of the stack pointer to fetch the least two significant digits of the return address from location 00FEH and loading them into the least significant digits of the program counter. The stack pointer is then incremented and its new contents used to fetch the two most significant digits of the program counter. The stack pointer is then incremented once more so that it contains 0100H. Although the RET instruction uses the contents of 00FEH and 00FFH without changing them, the ia7301 monitor "disturbs" these locations so that an examination of them at this time will no longer show the 0103H return address stored there.

It is important to keep straight the process of incrementing and decrementing the stack pointer. The monitor of the ia7301 automatically sets the stack pointer to 0100H whenever CLR is pressed. This remains unchanged until the computer executes a CALL instruction. The stack pointer is then decremented once BEFORE the return address is saved in memory. Thus the address will be saved at location 00FFH and, after a second decrementing operation, at 00FEH. The program beginning at location 0100H will not be disturbed. During the execution of the subroutine the stack pointer still points to location 00FEH so that, when the end of the subroutine is reached and the RET is executed, the first part of the return address is fetched from location 00FEH. Since the stack pointer was decremented twice during the execution of the CALL, it must now be incremented twice. This is necessary to return the stack pointer to the condition it enjoyed just prior to the CALL. As we shall see in future chapters, it is extremely important to maintain the contents of the stack pointer so that it is the same after exiting a subroutine as when it entered it. These points are illustrated in Figs. 5-6, 5-7, and 5-8.

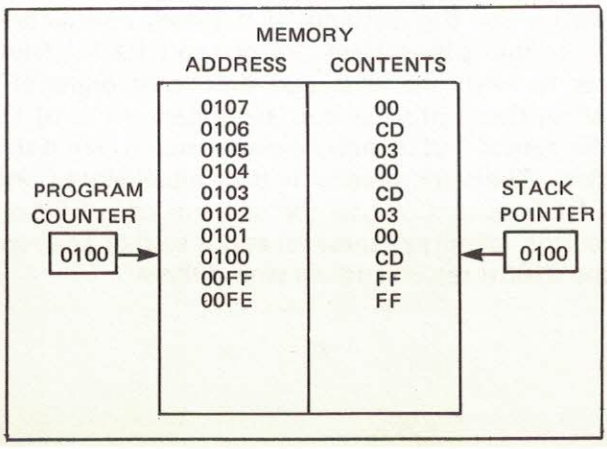


Fig. 5-6. Prior to executing the program, the stack pointer points to memory location 0100H. We have loaded locations 00FEH and 00FFH with FFH. This is done simply to make it easier to detect changes in the contents of these locations. When the first instruction, CALL 0300H, is executed the program counter will be loaded with the CALL address, 0300H, and program execution will continue at that point.

Before going on we should discuss one more aspect of the program we have been executing. During the first CALL of the subroutine, the subroutine executes properly because we initialized the D and E registers each to FFH prior to calling the subroutine. But how does the subroutine operate on the second and third CALL? After all, we do not initialize the registers prior to the second and third CALL. Well, actually we do. When the subroutine is completed the first time, the D and E registers both contain 00H. When the subroutine is called the second time, it is entered with these values already in the registers. Since the first instruction of the subroutine is a DCR E instruction, the E register is decremented by one prior to performing the JNZ test that sets up the loop. Decrementing a register set to 00H will result in it containing FFH. The situation is somewhat analagous to resetting an odometer to a car. You can turn it backwards to zero, but if you got too far the instrument will begin over at 9,999....9,998....9,997 and so on. Thus when the first JNZ test is encountered in the subroutine the E register contains FFH and

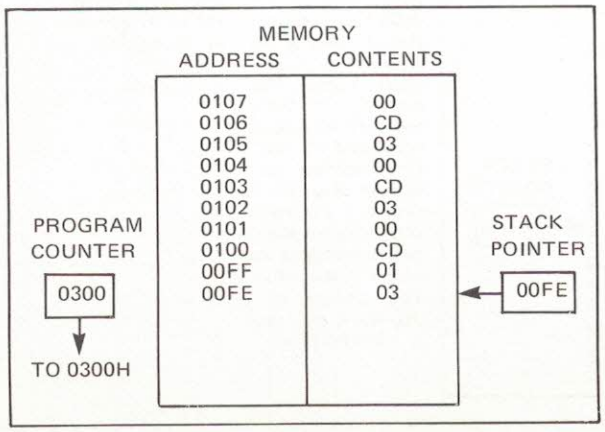


Fig. 5-7. When the CPU encounters the code for CALL, or CD, it causes the contents of the next two locations to be loaded into the program counter. The previous contents of the program counter are stored in memory at locations determined by the stack pointer. During this process the stack pointer is decremented twice. Notice that the data stored at locations 00FEH and 00FFH is actually the address of the next instruction in the main program stream.

the loop executes as normal. The same thing occurs with the D register. Although the subroutine requires 257 passes instead of 256 as it did on the first call, the difference is not noticeable.

Operating the LEDs. We just saw that the EXECUTE LED was lighted during the execution of the user's program. This is accomplished by the monitor program which causes the LED to be energized when the user's program is begun. At the conclusion of the program the monitor causes the LED to be extinguished. If the monitor can accomplish this, we should be able to duplicate the feat.

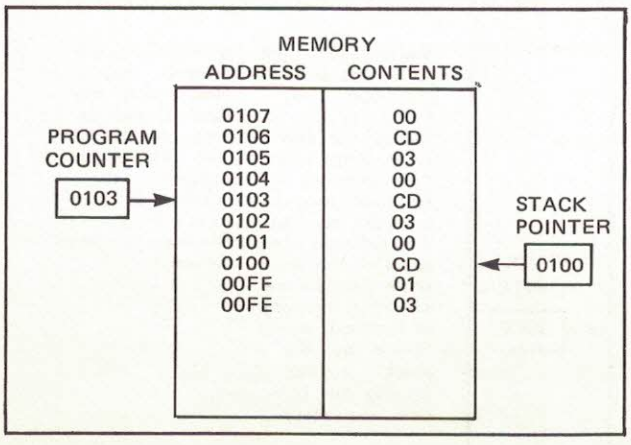


Fig. 5-8. When the RET instruction is found it signifies that the end of the subroutine has been reached, The contents of the memory locations addressed by the stack pointer are loaded into the program counter and program execution will pick up where it left off. The content of the stack pointer is incremented twice in the process. it is now ready to execute another CALL.

Since the ia7301 uses the memory-mapped I/O configuration for its I/O ports, memory addresses are assigned to each of its I/O devices. The LEDs in the lower-left corner of the system are assigned the address 6000H. By writing the appropriate data to this address we can make the LEDs respond to our program instead of the monitor program. Writing FBH to location 6000H will make the WRITE TAPE LED light, writing FDH to the same location will light the READ ~~EXECUTE~~ LED. We could do this by loading the accumulator with the appropriate data via the DCR mode and then using a STA 6000H instruction to write the contents of the accumulator into the memory location that drives the LEDs. There is a more elegant way.

One of the uses of microcomputers is to control actions and processes automatically without human intervention. To that end, we will seek to make our programs function without having to initialize registers and memory locations beyond the obvious task of loading the program. One way to do this is to make use of a class of instructions termed the Immediate instructions. For instance, Move Immediate to Accumulator, MVI A,D8, serves to load two hex digits of data into the accumulator. The instruction is called Immediate because it carries the very data needed by the instruction. MVI A, FBH, for instance, loads the accumulator with the data, FBH. Naturally this entire instruction cannot be stored in a single memory location. Two are required, one for the instruction and one for the data needed by the instruction. The hex code for the MVI A, D8 instruction is 3EH. That code must be followed by two hex digits that represent the data for the instruction. D8 is used here to symbolize the data; the significance of the numeral 8 that follows the letter D is tied to the concept of binary numbers and will be explored in Chapter 9.

```
0100    3E      MVI A, FBH      ;Load the accumulator with
0101    FB                      ;data FBH.
0102    32      STA     6000H   ;Store the data in the
0103    00                      ;memory location 6000H.
0104    60                      ;
0105    C3      JMP     0105    ;Jumpt to 0105H to set up an
```

```
0106    05          ;endless loop.
0107    01
```

Notice that the MVI A instruction requires two memory locations, 0100H for the instruction and 0101H for the data. The flow chart for the program is in Fig. 5-9. We load and executed the program as shown below.

Enter:	See Displayed:
DCM 0 1 0 0	0100---
NXT 3 E	0100-3E
NXT F B	0101-FB
NXT 3 2	0102-32
NXT 0 0	0103-00
NXT 6 0	0104-60
NXT C 3	0105-C3
NXT 0 5	0106-05
NXT 0 1	0107-01
NXT	0108-
CLR EXC	

Does the program work? We're used to seeing the regular displays come up in STEP; this time they went permanently blank. The EXECUTE LED did not go on, although the WRITE TAPE LED was lighted as was our plan. Actually the EXECUTE LED was lighted by the monitor in the process of beginning our program, but the first few instructions in our program overrode the monitor and extinguished it before the light was visible. You can see the EXECUTE LED light

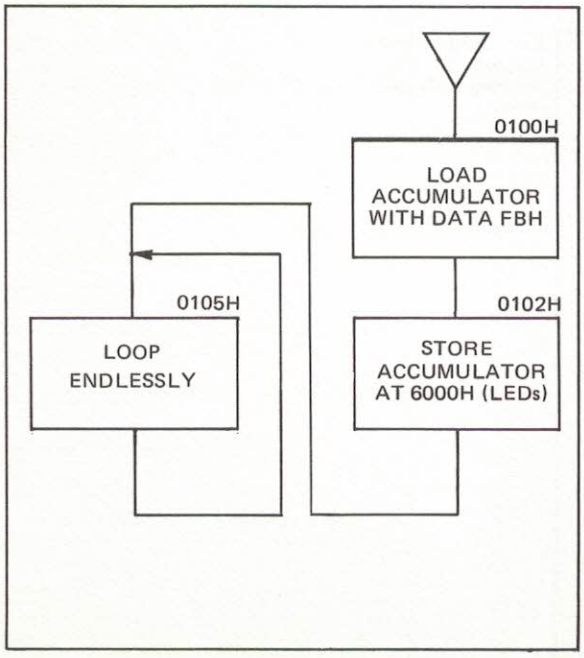


Fig. 5-9. Flow diagram for program to light WRITE TAPE LED

by pressing STEP a few times. Each time this key is depressed the EXECUTE LED will light during the time the key is held down. The first time this is done the monitor will extinguish the WRITE TAPE LED and it will not go back on without rerunning the program. This is because the STEP key causes one instruction to be executed, and since that instruction is a JMP to setting up an endless loop, the program is never reentered at a point that would relight WRITE TAPE.

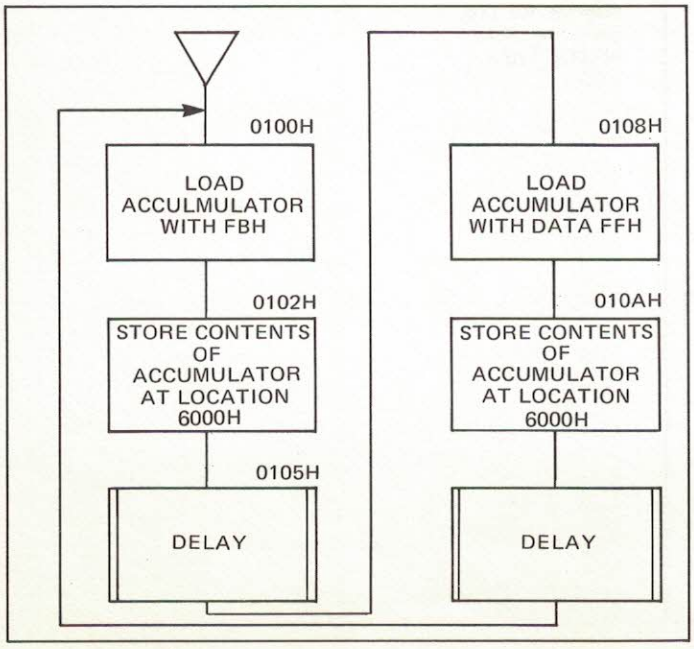


Fig. 5-10. Flow diagram of the flashing LED program.

Flashing LED. With only a little effort we can make the LED flash. Sending data FFH to location 6000H will extinguish all of the LEDs, so that by alternately sending FFH and FBH we can make the WRITE TAPE LED flash on and off. We will need some sort of timing action to delay the LED in each of the states so that we will be able to see it, but we have that subroutine still in our memory at location 0300H. Each time it is called a delay of about a half second is produced which should be just right for the flashing action. We will have to CALL it twice, once for the LED "off" state, but after some of the things we have done so far that should'nt be a problem. The program is summarized below and in the flow diagram of FIG. 5-10.

```
0100    3E    MVI A, FBH      ;Store data FBH in accumulator
0101    FB                    ;to light WRITE TAPE LED.
0102    32    STA    6000H    ;Store contents of accumulator
0103    00                    ;at location 6000H.
0104    60                    ;
0105    CD    CALL   300H     ;CALL DELAY subroutine.
0106    00                    ;
0107    03                    ;
0108    3E    MVI A, FFH      ;Store data FFH in accumulator to
0109    FF                    ;extinguish WRITE TAPE LED.
010A    32    STA    6000H    ;Store contents of accumulator
010B    00                    ;at location 6000H.
010C    60                    ;
010D    CD    CALL   300H     ;CALL DELAY subroutine.
010E    00                    ;
010F    03                    ;
0110    C3    JMP    0100H    ;Return to the beginning of the
0111    00                    ;program so that it repeats.
0112    01                    ;
```

```
0300    1D      DCR     E       ;Delay Subroutine.
0301    C2      JNZ     0300H   ;
0302    00                      ;
0303    03                      ;
0304    15      DCR     D       ;
0305    C2      JNZ     0300H   ;
0306    00                      ;
0307    03                      ;
0308    C9      RET             ;
```

We load the program as shown below. If you load it correctly the results when you execute it should be immediately apparent.

Enter:	See Displayed:
CLR 0 1 0 5 NXT	0105-C3
C D	0105-Cd
NXT 0 0	0106-00
NXT 0 3	0107-03
NXT 3 E	0108-3E
NXT F F	0109-FF

NXT 3 2	n010A-32
NXT 0 0	n010b-00
NXT 6 0	n010C-60
NXT C D	n010d-Cd
NXT 0 0	n010E-00
NXT 0 3	n010F-03
NXT C 3	n0110-C3
NXT 0 0	n0111-00
NXT 0 1	n0112-01
NXT	n0113-
CLR EXC	

At this point your WRITE TAPE LED should be flashing away like crazy. If it isn't, better go back and check to be sure the program is loaded correctly.

With only a slight variation we can introduce another LED into the program above, so that the two are flashing alternately. Instead of blanking all of the LEDs with the MVI A, FFH instruction we can cause the WRITE TAPE to be extinguished and the READ TAPE LED lighted by changing the data word in that instruction to FDH.

Enter:	See Displayed:
CLR 0 1 0 9 NXT	
F D	
NXT CLR EXC	

This should cause the WRITE TAPE LED to be lighted for a half second, then WRITE TAPE should go out and READ TAPE go on. This action should be repeated as long as you care to watch.

Indirect Addressing. Before going on, let's try to find some ways to streamline our program somewhat. If we were to keep sending data to location 6000H to make the LEDs flash in some sort of pattern, we would find that the use of the STA 6000H instruction was taking up a lot of memory. After all, it requires three memory locations, one for the instruction and two for the address 6000H. It seems inefficient to keep repeating the address over and over again.

The answer is in a concept called Indirect Addressing. Basically it uses another set of registers to hold the address, and then an instruction using only one memory location can be used to access memory. One example of this is the Store Accumulator instruction, STAX. There are two STAX instructions, but the one we will use is STAX B. This instruction causes the contents of the accumulator to be stored at a memory location whose address is contained in the B and C registers. Why C? Because an address requires four hex digits to specify and the normal working registers of the CPU hold only two. Two together, however, can hold four hex digits and hence, a memory address. In this scheme, the B register contains the two most significant hex digits and the C register the two least significant digits. By loading B with 60H and C with 00H we can use a STAX B instruction to send the contents of the accumulator to address 6000H.

If this seems like a lot of bother just to send data to a memory location, you'd be right! With only two occasions to use that particular memory location we'd be much further ahead with our original program. However the indirect addressing method has a lot of versatility built into it that our original methods did not. Stick with us; there's gold in those registers!

One more thing. In keeping with our earlier resolve to use the program to do all the initializing as much as possible, (instead of cheating with the DCR key), we'll have to use instructions to load the B and C registers with the appropriate values. Just as there is a MVI A, D8 instruction to load data into the accumulator, so are there MVI B, D8 and MVI C, D8 instructions to load data into the B and C registers.

Let's try putting all of these instructions together to duplicate the program for flashing the LEDS as before. Once done, we can use the new instructions to expand the programs. To do this we will use the system of flow diagrams with which we have been illustrating the text. This is the preferred way of writing any program since it puts the problem into simple graphic terms that are easily seen and grasped.

We begin by initializing the B and C registers using the instructions just mentioned. Of course, in the flow diagram it will not be necessary to detail exactly which instructions are to be used to accomplish the tasks set forth in the boxes. Indeed, as our programs become more complex, the operations represented by the boxes will require many instructions. Once the B and C registers are initialized to address 6000H, the accumulator will be loaded with the data FBH to light the WRITE TAPE LED. This will be followed by an operation to move the contents of the accumulator to address 6000H which causes the actual lighting of the LED. The subroutine DELAY must then be called to make the action of the LED visible. This is followed by loading the accumulator with the data FDH to light the READ TAPE LED and move it to 6000H. All that remains is to jump back to location 0100H to make the program repeat itself. See Fig. 5-11.

The next step is to list the program steps location by location so they can be loaded into the memory. The hex codes for the three new instructions are listed below.

Mnemonic	Hex Code
MVI B, D8	06H
MVI C, D8	0EH
STAX B	02H

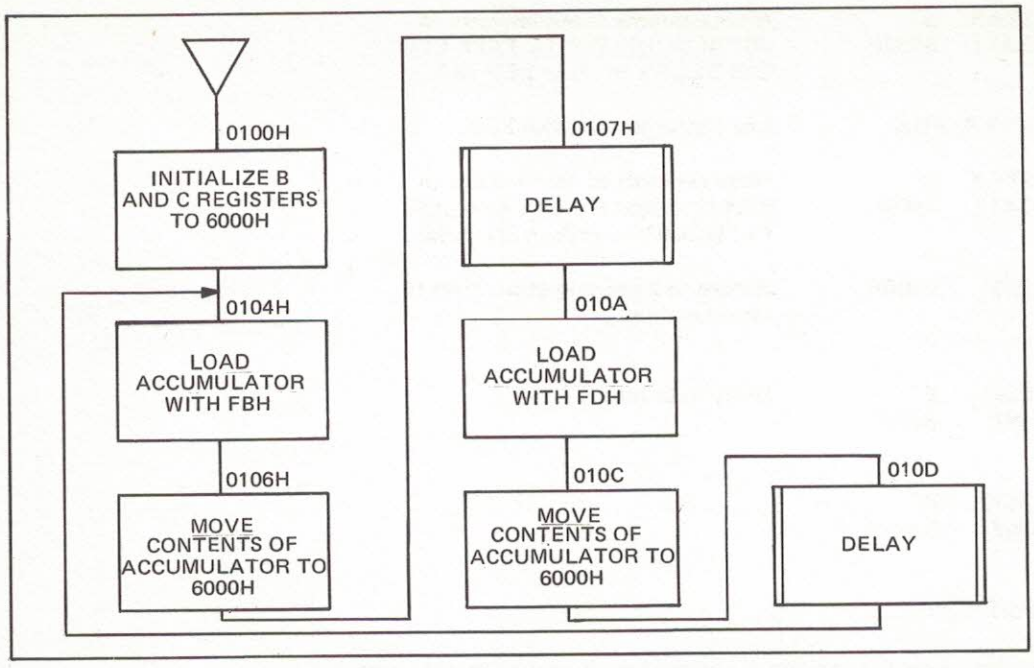


Fig. 5-11. Flow diagram of program to alternately flash LEDs using indirect addressing.

```
0100    06      MVI B, 60H      ;Load B and C registers with the
0101    60                      ;memory address 6000H.
0102    0E      MVI C, 00H      ;
0103    00                      ;
0104    3E      MVI A, FBH      ;Load accumulator with FBH.
0105    FB                      ;
```

```
0106    02      STAX    B       ;Move contents of accumulator to
0107    CD      CALL    0300H   ;6000H to light WRITE TAPE LED.
0108    00                      ;Call DELAY to make LED visible.
0109    03                      ;
010A    3E      MVI A, FDH      ;Load accumulator with FDH.
010B    FD                      ;
010C    02      STAX    B       ;Move contents of accumulator to
010D    CD      CALL    0300H   ;6000H to light READ TAPE LED.
010E    00                      ;Call DELAY to make LED visible.
010F    03                      ;
0110    C3      JMP     0100H   ;Return to beginning of program to
0111    00                      ;repeat action.
0112    01                      ;

0300    1D      DCR     E       ;Delay Subroutine.
0301    C2      JNZ     0300H   ;
0302    00                      ;
0303    03                      ;
0304    15      DCR     D       ;
0305    C2      JNZ     0300H   ;
0306    00                      ;
0307    03                      ;
0308    C9      RET             ;
```

Notice that this requires exactly the same amount of memory as our original.

All that remains is to load the program from our listing.

Enter:	See Displayed:
CLR	
0 1 0 0 NXT	
0 6	
NXT 6 0	
NXT 0 E	
NXT 0 0	
NXT 3 E	
NXT F B	
NXT 0 2	
NXT C D	
NXT 0 0	
NXT 0 3	
NXT 3 E	
NXT F D	
NXT 0 2	

NXT C D	
NXT 0 0	
NXT 0 3	
NXT C 3	
NXT 0 0	
NXT 0 1	
NXT	

Rather than executing the program using EXC , let's step our way through and examine the action of the various registers as the instructions load them and move data around.

In order to see the changes take place in the registers, let's write the contents of each using the DCR mode.

Enter:	See Displayed:	Write Contents:
DCR		_____
NXT NXT		_____
NXT		_____

Having recorded the contents of the registers in question, press CLR to reset the program counter and stack pointer, and STEP the first instruction.

Enter:	See Displayed:	Write Contents:
CLR STEP	F 0 102 - 0E	
DCR NXT NXT	B - - - - - 60	_____

The MVI B, 60H instruction at location 0100H has caused the register B to be loaded with 60H. In our earler programs we had to load and initialize the registers using the DCR mode. Now the program does it for us! STEP the next instruction, the MVI C, 00H and check the register with the DCR key.

Enter:	See Displayed:	Write Contents:
STEP	F 0 104 - 3E	
DCR NXT NXT NXT	C - - - - - 00	_____

The MVI C, 00H instruction worked as well. The next depression of STEP should load the accumulator with the data FBH.

Enter:	See Displayed:	Write Contents:
STEP	F 0 106 - 02	
DCR	A - - - - FB	_____

Apparently the MVI instructions operate as described. You will find this block of instructions very handy for loading data into the various working registers of the CPU. In the coming chapters we will use them over and over again to initialize the various registers and even memory

locations. Now though, let's move on and see STAX B in operation. This instruction will cause the contents of the accumulator to be stored at address 6000H. That location is the address of the LEDs, and storing FBH there will cause the WRITE TAPE LED to light. Because this instruction only requires one memory location in the program, it is very useful when frequent operations to the same location are required.

Enter:	See Displayed:
STEP	

Shouldn't we see the WRITE TAPE LED light up at this point? Granted, we're sitting still in the program, frozen in the STEP mode to display location 0107H and its contents, but doesn't that have the same effect as the DELAY subroutine which is about to be called by the program anyway? Ah, but don't forget the monitor program. The CPU in the computer is always running, executing some program. What appears to you as a moment captured through magic of STEP is, in fact, a portion of the monitor which displays the contents of a memory location and its address. In the process of doing that, the monitor blanked the LEDs effectively overriding your instruction to light WRITE TAPE. Apparently we will only be able to see the action of the LEDs when we execute the program in the free-running mode via EXC . In the meantime, pressing STEP again will only take us into the DELAY subroutine with its 256 loops, and since we've already examined that, let's reset the program counter and stack pointer and really try the program.

Enter:	See Displayed:
CLR EXC	

Just as before, our two LEDs are doing their trick. The concept of indirect addressing is so powerful that there are a whole set of instructions that make use of this technique. We'll explore them in good time, but first we want to talk about another way of loading the registers. Because indirect addressing is used so often, we will find ourselves loading addresses into the various registers of the CPU very frequently. As has been pointed out, addresses require two registers to hold the four hex digits, and using the MVI instructions just discussed requires two instructions to load the address into the registers.

When the registers are used to hold addresses they must be treated as pairs, and the CPU has been set up so that the registers are already somewhat married to one another to facilitate this pair-relationship. Accordingly the B and C registers are used as a pair, the D and E registers form a pair, and the H and L registers make up a pair. They can still be used individually, mind you, but for the purpose of holding addresses, or any other four digit number for that matter, the registers will be used on a pair basis. Thus STAX B uses the address in the B and C registers as a destination for the contents of the accumulator. By the same token, STAX D uses D and E registers to form an address for the same purpose.

In order to facilitate the loading of addresses and other four digit numbers into the register pairs, a special class of instructions have been provided that operate on both registers at once. One of these is LXI B, Load Immediate Register Pair B, and it causes four digits of data to be loaded into the B and C registers. This instruction requires three memory locations, one for the instruction and two for the data. This represents a savings of one memory location over the use of MVI B and MVI C in combination each of which require two locations. The hex code for LXI B, is 01H; it must always be followed by four hex digits which will be loaded into the register pair.

We can ammend our program by replacing the two MVI instructions with the single LXI B instruction. Since this occurs at the beginning of the program it will necessitate rippling all of the instructions that follow it to make the program correct.

```
0100    01    LXI B, 6000H      ;Load B and C registers with the
0101    00                      ;memory address 6000H.
0102    60                      ;
0103    3E    MVI A, FBH        ;Load accumulator with FBH.
0104    FB                      ;
0105    02    STAX   B          ;Move contents of the accumulator
0106    CD    CALL   0300H      ;to 6000H to light WRITE TAPE LED.
0107    00                      ;Call DELAY to make LED visible.
0108    03                      ;
0109    3E    MVI A, FDH        ;Load accumualtor with FDH.
010A    FD                      ;
010B    02    STAX   B          ;Move contents of accumulator to
010C    CD    CALL   0300H      ;6000H to light READ TAPE LED.
010D    00                      ;Call DELAY to make LED visible.
010E    03                      ;
010F    C3    JMP    0100H      ;Return to beginning of program to
0110    00                      ;repeat action.
0111    01                      ;

0300    1D    DCR    E          ;Delay Subroutine.
0301    C2    JNZ    0300H      ;
0302    00                      ;
0303    03                      ;
0304    15    DCR    D          ;
0305    C2    JNZ    0300H      ;
0306    00                      ;
0307    03                      ;
0308    C9    RET               ;
```

Enter:	See Displayed:
CLR 0 1 0 0 NXT	
0 1	
NXT 0 0	
NXT 6 0	
NXT 3 E	
NXT F B	
NXT 0 2	
NXT C D	
NXT 0 0	
NXT 0 3	
NXT 3 E	
NXT F D	
NXT 0 2	
NXT C D	
NXT 0 0	

NXT 0 3	
NXT C 3	
NXT 0 0	
NXT 0 1	
NXT	

Since the program loads all of the registers itself, without having to resort to the DCR key, we can execute the program just by pressing EXC .

Enter:	See Displayed:
CLR EXC	

You should see the LEDs flash in the same fashion as they did earlier, proving that the LXI B instruction works the same way as the two MVI instructions. If you compare the length of this program with the earlier versions you'll find that it requires one less location to store it. Saving one location may not seem important, but it could amount to a hundred locations when the savings occurs over and over again in a major program. And since loading register pairs in this manner is one of the most common program steps, those kinds of memory savings are real. Once the LXI B instruction has been used to set up the registers, the STAX B instruction can be used to move data to the LEDs over and over again without having to reinitialize the registers.

Using the Display LEDs. So far we have been discussing the three discrete LEDs in the corner of the board that provides a monitor function indicating EXECUTE, WRITE TAPE and READ TAPE. Although we did not mention it at the time, these LEDs use a register to save the data that we write to them at location 6000H. That way when we write data to the LEDs, the register accepts the data and stores it. If it were not for this register, the data would disappear as soon as the next instruction was executed. Because of the register, we can write data to the LEDs once, and that combination of LEDs will light and remain lighted until new data is sent to the register. In this sense, the LEDs are similar to many other output ports that feature registers as part of the port. These ports are called latching output ports and reduce the amount of programming steps necessary to interface to many types of peripherals.

The seven-segment LEDs that display memory addresses and their contents are somewhat different. These use a circuit configuration common to LED displays called a multiplexing drive circuit. This type of circuit saves components but requires a more involved program to drive the LEDs. Basically the multiplexing technique scans the LEDs so that only one of the display digits is on at a time. With eight displays, each digit is energized approximately 12% of the time. This would normally reduce the brightness of the displays considerably, but the current limiting resistors that control the amount of current through the display have been reduced so that the current is increased. The increased current causes the displays to brighten by exactly the same amount the reduced duty cycle dimmed them. The end result is that the displays are nice and bright and a lot less parts are needed. That's the good news.

The bad news is that the savings in hardware makes the programming more difficult. The program must be written so that it energizes one and only one of the displays at a time. In effect we are scanning the displays under program control. Of course it is not enough to merely energize the displays. A single register is used for all of the data for all eight of the displays. Since the register can only hold the data for one of the displays at a time, the program must rotate the data

at the same time it is scanning the displays so that the correct data is loaded into the register prior to the correct display digit being energized. That way the right data will meet the right digit at the right time. If everything comes together correctly, all of the digits will be lighted, each displaying the hex digit appropriate to its location.

This is a little too involved to just jump into all at once (besides, there's a few other little twists that we haven't described yet). Instead, we're going to work out the problems by doing parts of the whole program and then tie the segments together to get a complete program that works.

Let's start by just trying to get one LED to light. Then we can expand the program to include all the displays. The address of the register that stores the display data is CXXXH. We can write data into this memory location just like any other place in memory. If we then energize one of the display LEDs, the data in the display register will appear in the energized LED. Sound simple? There's just one more thing. The LED will only stay on for about one millisecond (that's one one-thousandth of a second), because there is a special circuit that turns the LED off right after you turn it on. Why do such a counterproductive thing? Because with the multiplexing of the LEDs with each LED carrying a very large current for only 12% of the time, there is a potential problem. If our program correctly scans the LEDs so that each is truly on for just a brief period, everything is fine. But if for any reason, the LED is turned on steady, 100% of the time, that high current will permanently damage the display. To prevent that from happening, we have incorporated a special guard circuit in the LED drives so that they are automatically turned off after one millisecond. Now that's a much shorter period of time than you can see, and if we stopped here, the LEDs would never be visible. What we must do is keep writing the data to the LEDs over and over again. That way the guard circuit turns the LED off and we turn it back on with another write data operation. Of course, we have to be careful that we don't turn it on until after the guard circuit turns the display off. You see, the guard circuit can be fooled by just continually writing data to the LED. Each time it receives data, the guard circuit is reset to

start its one millisecond timing period again. If we are writing data to it faster than the guard circuit can complete its timing operation, the guard circuit will never shut off the LED, and we will have defeated the guard action and eventually burn the LED out.

To prevent this we will have to have our program monitor the guard circuit so that data is not re-written to the LED until the 0.2 millisecond is up. That way the guard circuit will be reset and the timing action will occur correctly. At the same time we have to let the LED stay off for a while before we send data to it again. This is to let it cool down before hitting it with another high-current pulse. That is done automatically when we are scanning all of the displays since each LED is cooling while the other seven displays are being energized as their turn comes up. But when only one display is going to be used, we'll have to be a little more careful.

For a first shot at this problem, let's try to get the ~~right~~ Left-most LED to light. Never mind what's going to appear; we'll take care of that in the next chapter. For now, we'll be satisfied if we can just get the displays to light. The first step is to write some data into the display register. Since this is a latching type of output port, same as the three discrete LEDs in the corner of the board, the data will remain in the register until the guard circuit shuts it off. The address of that port is CXXXH. The X's here indicate that only the first of the four hex digits matters; if the first digit is CH, the rest can be anything and the register will still be correctly addressed. We can use a STAX D instruction for this; of course, the D and E registers will have to be loaded with address CXXXH. Since only the most significant of the four address digits matters, it should be easy to get this part of the address into the D register. Since the other three digits can be anything, we won't have to load the E register per se; the program will execute correctly with whatever happens to be there when we turn the computer on. The data that is sent to this register will be whatever is in the accumulator, and since we will want to do some experiments with various data numbers appearing in the displays, it would be helpful to load a known quantity into the accu-

mulator with a MVI A instruction. Let's try to get a zero the appear. That should take a zero in the accumulator so we'll make our second instruction MVI A, 00H.

The same register address that holds the display data also addresses a second register that holds a data word that energizes one of the LED display digits. This is a register that, like the others we have been discussing accepts its data and latches it into the register whenever a write operation to that address occurs. Unlike previous registers, however, this register accepts its data from the address bus and not the data bus. We have already seen that the address of the display data register is CXXXH. This address will also send data to the digit enable register which enables the various display digits. Within this register are electronic circuits that select one of the eight display digits and enable it. These circuits utilize the second most significant digit to select which of the displays will be enabled. This, in effect, assigns addresses to the individual display digits. These addresses are shown below.

CFXXH	D_7	**(left-most digit display)**
CDXXH	D_6	
CBXXH	D_5	
C9XXH	D_4	
C7XXH	D_3	
C5XXH	D_2	
C3XXH	D_1	
C1XXH	D_0	**(right-most digit)**

The fact that this register accepts its data off the address bus and not the data bus means that any memory write operation to one of these addresses, regardless of what data is being sent over the data bus, will enable one of the displays. And since the memory write operation that sent data to the display register did not make use of any but the most significant address digit, we can combine the two operations into one.

Since we wish to have data appear in the ~~right~~ left-most display, if we send it to memory location CFXXH not only will we load the data register with the data but at the same time enable the correct display digit. This is done with a STAX D instruction, where D/E have been previously loaded with the address CFXXH. This will cause the right-most digit to light up with the contents of the accumulator.

Before we can continue, we should be sure the guard circuit has completed its timing operation. Although it is possible to actually have the program monitor the guard circuit, at this time it is much simpler to simply set up a short delay loop similar to those we have already been using. Since the B register has not yet been used by our program we can use it for the loop. This is done by loading it with a timing constant, 4FH, decrementing it, testing to see if it is zero, and, if not zero, returning to the decrement operation. This loop is illustrated in Fig. 5-12.

Once the timing loop, referred to as DELAY 1, has been completed we can move on. Normally this would be done by going on to the next display digit, but since we're only going to be using one digit, we'll have to fake this so that the LED digit operates on the one-eighth duty cycle for which it was designed. This is done by setting up the C register as a short loop counter. It calls DELAY 1 a total of seven times, which should provide the single display we're using with about the right duty cycle. This second program is called DELAY 2 and is illustrated in Fig. 5-13. The whole program is illustrated in Fig. 5-14 and below.

```
0100    16      MVI D, CFH      ;Load D register with first half of
0101    CF                      ;address CFXXH.  E can be ignored at
0102    3E      MVI A, 00H      ;at this time since only two digits
0103    00                      ;of address are used.  Load accumu-
0104    12      STAX   D        ;lator with data 00H and write to
0105    CD      CALL DELAY 1    ;data register.  Delay.
0106    00                      ;
0107    03                      ;
0108    CD      CALL DELAY 2    ;Delay to correct duty cycle for
0109    00                      ;other seven displays not being used.
010A    02                      ;
010B    C3      JMP    0104H    ;Jump back to 0104H to close loop.
010C    04                      ;This way display will remain lighted.
010D    01                      ;

0300    06      MVI B, 4FH      ;Load register B with timing constant.
0301    4F                      ;This value has been chosen to simulate
0302    05      DCR    B        ;the guard circuit timing out.
0303    C2      JNZ    0302H    ;If B is not zero, go back and decrement
0304    02                      ;again.  If zero, return to calling
0305    03                      ;program.
0306    C9      RET

0200    0E      MVI C, 07H      ;Set up register C as loop counter.
0201    07                      ;
0202    CD      CALL DELAY 1    ;Delay 1 simulates each of the other
0203    00                      ;digits.  The loop counter causes
0204    03                      ;this to happen seven times.
0205    0D      DCR    C        ;Decrement loop counter.
0206    C8      RZ              ;If zero, return to the calling
0207    C3      JMP    0202     ;program.  If not go back to 0202H.
0208    02                      ;
0209    02                      ;
```

Study this program carefully. Notice that DELAY 1 is used twice, once by itself and once within another subroutine, DELAY 2. Notice also that there are some new instructions appearing here for the first time. Most of these should be easy to understand. MVI D, D8, hex code 16H, causes the two hex digits of data, D8, to be loaded into the register D. This instruction is just like the MVI B, D8 and MVI C, D8 that we have already used. The STAX D instruction, hex code 12H, is just like STAX B, except that the contents of the accumulator are stored at the memory location addressed by the D/E register pair instead of the B/C register pair. We have previously decremented the D and E registers with DCR D and DCR E, and now we shall decrement the B and C registers with DCR B, hex code 05H, and DCR C, hex code 0DH, respectively.

The RZ instruction might not be so clear. Return if Zero, hex code C8H, causes the return to the calling program if and only if the zero flag is set indicating that the last operation resulted in a zero. This is very similar to the conditional jump, Jump if not Zero, that we have already used. If the zero flag has not been set, the RZ instruction is ignored and program execution goes on to the next instruction in the numerical program sequence. Since that instruction is a jump back to 0202H, a loop is set up and the program execution will go around the loop until decrementing the C register causes it to contain zero. At that time the loop stops and the RZ instruction causes program execution to return to the calling program.

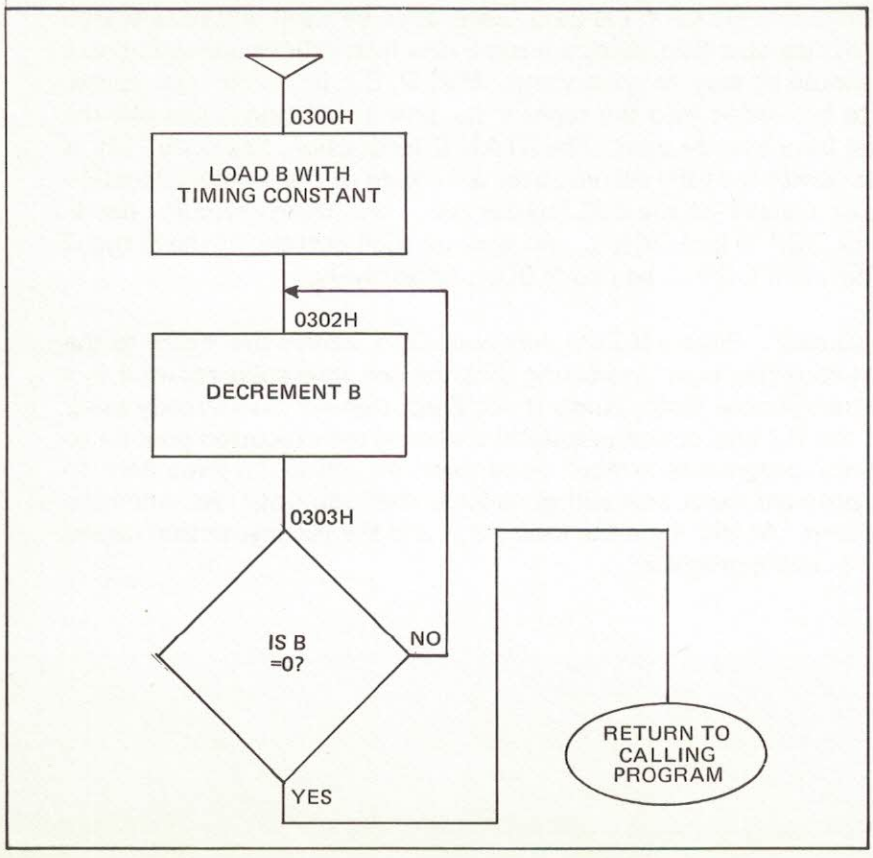


Fig. 5-12. This timing loop, DELAY 1, is set up to simulate the length of time one of the display digits would be lighted. The timing constant, 4FH, has been chosen so that the time to execute the entire subroutine is about the same as the timing period of the guard circuit.

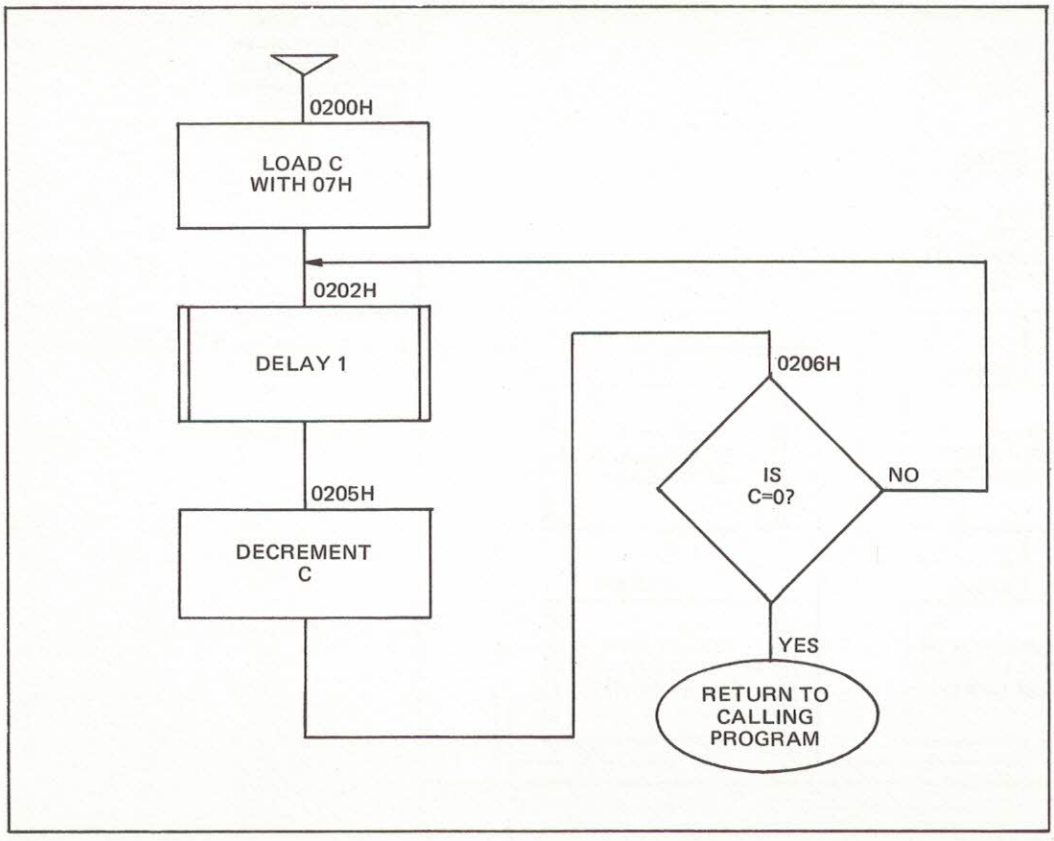


Fig. 5-13. DELAY 2 causes DELAY 1 to be CALLed seven times to simulate the time that the other seven digits would be lighted.

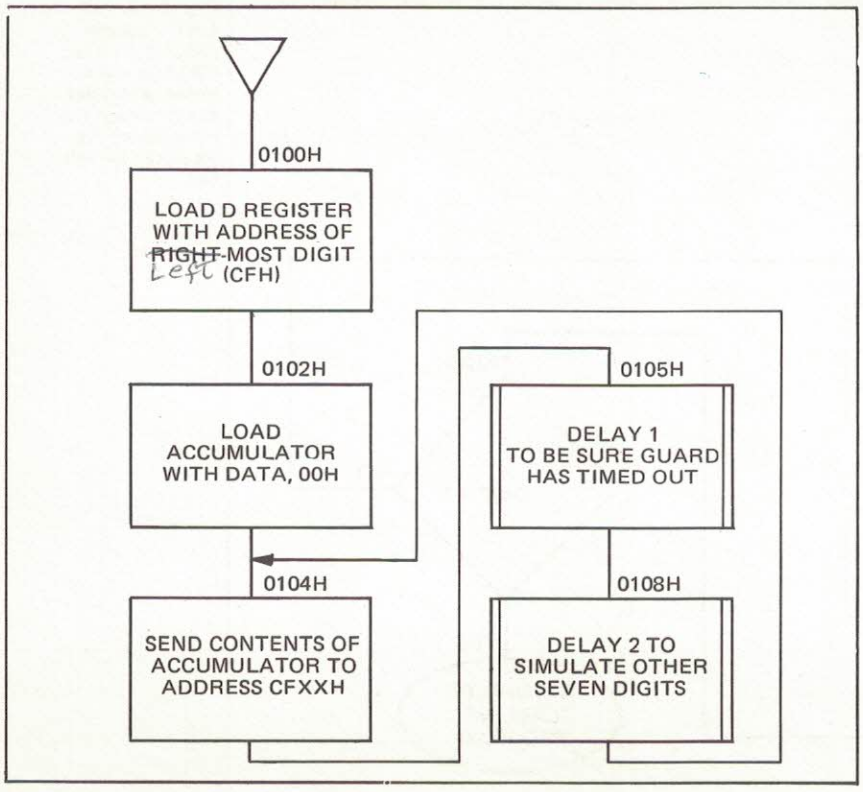


Fig. 5-14. Complete flow-diagram of the program to light the ~~right~~ Left-most display.

We load the program below:

Enter:	See Displayed:
DCM 0 1 0 0 NXT	
1 6	
NXT C F	
NXT 3 E	
NXT 0 0	
NXT 1 2	
NXT C D	
NXT 0 0	
NXT 0 3	
NXT C D	
NXT 0 0	
NXT 0 2	
NXT C 3	
NXT 0 4	

NXT 0 1	n0100d-01
NXT	n0100E-??
DCM 0 2 0 0 NXT	n0200-??
0 E	n0200-0E
NXT 0 7	n0201-07
NXT C D	n0202-Cd
NXT 0 0	n0203-00
NXT 0 3	n0204-03
NXT 0 D	n0205-0d
NXT C 8	n0206-C8
NXT C 3	n0207-C3
NXT 0 2	n0208-02
NXT 0 2	n0209-02
NXT	n020A-??
DCM 0 3 0 0 NXT	n0300-??
0 6	n0300-06

NXT 4 F	
NXT 0 5	
NXT C 2	
NXT 0 2	
NXT 0 3	
NXT C 9	
NXT	

We can execute the program by pressing CLR to reset the program counter and stack pointer and the EXC .

Enter:	See Displayed:
CLR EXC	

The program for lighting the display digit works but apparently loading a zero into the accumulator does not produce a zero in the display. Repeat the experiment using 55H in the accumulator (by changing the data in the MVI A instruction at locations 0102H and 0103H).

Enter:	See Displayed:
CLR DCM 0 1 0 2 NXT	0102-3E
NXT 5 5	0103-55
NXT CLR EXC	

This proves that it is the contents of the accumulator that are determining what is displayed, but we can not simply take the data in the accumulator and send it directly to the display data register. The problem is caused by the fact that seven-segment displays have their own code. That is, 00H will not produce a zero in the display, 01H will not produce a one in the display, and so on. We need to have a table of code conversions and then have the computer look our data up in the conversion table, and send the new, converted data to the displays so we can read them in the number system with which we are familiar. This is a fascinating problem and will lead us to all sorts of new twists in our programming. Before we dive into that though, we want to take a moment and get the various instruction groups organized in our minds. Those groups are illustrated on the hex card that is at the front of your binder. Once familiar with that card you will be able to write programs yourself by looking up the hex codes for your instructions on the hex card.